Constructors:

A *constructor* initializes an object when it is created. It has the same name as its class and is syntactically similar to a method. However, constructors have no explicit return type. Typically, you will use a constructor to give initial values to the instance variables defined by the class, or to perform any other startup procedures required to create a fully formed object.
All classes have constructors, whether you define one or not, because Java automatically provides a default constructor that initializes all member variables to zero. However, once you define your own constructor, the default constructor is no longer used.
Here is a simple example that uses a constructor:

```
// A simple constructor.
class MyClass {
 int x;
  MyClass() {
   x = 10;
  }
}
class ConsDemo {
public static void main(String args[]) {
MyClass t1 = new MyClass();
MyClass t2 = new MyClass();
System.out.println(t1.x + " " + t2.x);
}
}
```

In this example, the constructor for **MyClass** is

```
MyClass() {
  x = 10;
}
```

This constructor assigns the instance variable **x** of **MyClass** the value 10. This constructor is called by **new** when an object is created. For example, in the line

```
MyClass t1 = new MyClass();
```

the constructor **MyClass( )** is called on the **t1** object, giving **t1.x** the value 10. The same is true for **t2**. After construction, **t2.x** has the value 10. Thus, the output from the program is

```
10 10
```

Parameterized Constructors

In the preceding example, a parameter-less constructor was used. Although this is fine for some situations, most often you will need a constructor that accepts one or more parameters.

Parameters are added to a constructor in the same way that they are added to a method: justndeclare them inside the parentheses after the constructor's name. For example, here, **MyClass** is given a parameterized constructor:

```
// A parameterized constructor.
class MyClass {
int x;
MyClass(int i) {
x = i;
}
}
class ParmConsDemo {
public static void main(String args[]) {
MyClass t1 = new MyClass(10);
MyClass t2 = new MyClass(88);
System.out.println(t1.x + " " + t2.x);
}
}
```

The output from this program is shown here:

```
10 88
```

In this version of the program, the **MyClass( )** constructor defines one parameter called **i**, which is used to initialize the instance variable, **x**. Thus, when the line `MyClass t1 = new MyClass(10);` executes, the value 10 is passed to **i**, which is then assigned to **x**. This constructor has a parameter.

Adding a Constructor to the Vehicle Class

We can improve the **Vehicle** class by adding a constructor that automatically initializes the **passengers**, **fuelcap**, and **mpg** fields when an object is constructed. Pay special attention to how **Vehicle** objects are created.

```
// Add a constructor.
class Vehicle {
int passengers; // number of passengers
int fuelcap; // fuel capacity in gallons
int mpg; // fuel consumption in miles per gallon

// This is a constructor for Vehicle.
Vehicle(int p, int f, int m) {
passengers = p;
fuelcap = f;
mpg = m;
}

// Return the range.
```

```
int range() {
return mpg * fuelcap;
}
// Compute fuel needed for a given distance.
double fuelneeded(int miles) {
return (double) miles / mpg;
}
}
class VehConsDemo {
public static void main(String args[]) {
// construct complete vehicles
Vehicle minivan = new Vehicle(7, 16, 21);
Vehicle sportscar = new Vehicle(2, 14, 12);
double gallons;
int dist = 252;
gallons = minivan.fuelneeded(dist);
System.out.println("To go " + dist + " miles minivan needs " +
gallons + " gallons of fuel.");
gallons = sportscar.fuelneeded(dist);
System.out.println("To go " + dist + " miles sportscar needs " +
gallons + " gallons of fuel.");
}
}
```

Both **minivan** and **sportscar** are initialized by the **Vehicle( )** constructor when they are created. Each object is initialized as specified in the parameters to its constructor. For example, in the following line,

```
Vehicle minivan = new Vehicle(7, 16, 21);
```

The values 7, 16, and 21 are passed to the **Vehicle( )** constructor when **new** creates the object. Thus, **minivan**'s copy of **passengers**, **fuelcap**, and **mpg** will contain the values 7, 16, and 21, respectively. The output from this program is the same as the previous version.

Controlling Access to Class Members:

In essence, there are two basic types of class members: public and private. A public member can be freely accessed by code defined outside of its class. This is the type of class member that we have been using up to this point.

A private member can be accessed only by other methods defined by its class. It is through the use of private members that access is controlled.

Restricting access to a class's members is a fundamental part of object-oriented programming because it helps prevent the misuse of an object. By allowing access to private data only through a well-defined set of methods, you can prevent improper values from being assigned to that data—by performing a range check, for example. It is not possible for code outside the class to set the value of a private member

directly. You can also control precisely how and when the data within an object is used. Thus, when correctly implemented, a class creates a "black box" that can be used, but the inner workings of which are not open to tampering.
Up to this point, you haven't had to worry about access control because Java provides a default access setting in which the members of a class are freely available to the other code in your program. (Thus, the default access setting is essentially public.) Although convenient for simple classes (and example programs in books such as this one), this default setting is inadequate for many real-world situations. Here you will see how to use Java's other access control features.

Java's Access Specifiers:

Member access control is achieved through the use of three access specifiers: **public**, **private**, and **protected**. As explained, if no access specifier is used, the default access setting is assumed.
When a member of a class is modified by the public specifier, that member can be accessed by any other code in your program. This includes methods defined inside other classes.
When a member of a class is specified as **private**, that member can be accessed only by other members of its class. Thus, methods in other classes cannot access a private member of another class.
The default access setting (in which no access specifier is used) is the same as public unless your program is broken down into packages. A package is, essentially, a grouping of classes.
An access specifier precedes the rest of a member's type specification. That is, it must begin a member's declaration statement. Here are some examples:

Constructor and Set and Get Methods:

Constructor Used to initialize an object of the class and any members. Called only once for the lifetime of the object being initialized:

```
public class Person
{
   String person_name;

      public Person(String name)
   {
      person _name = name;
   }
}
```

Example usage:

```
// Initialize object of Person class by calling constructor.
Person p = new Person("John Smith");
```

Get and Set Methods

Used to get or set values of object members respectively. Unlike constructors, set methods can be used to initialize member values more than once:

```
public class Person
{
    // Member.
    String person_name;
    // Constructor.
   public Person(String name)
   {
      person_name = name;
   }

    // Get member value.
    public String getName()
    {
       return person_name;
    }

    // Set member value to given value.
    public void setName(String name)
    {
       person_name = name;
    }
}
```

Example usage:

```
// Initialize object of Person class.
Person p = new Person("John Smith");
String name;
// Get name of this person.
name = p.getName();
System.out.println(name);

// Set new name for this person.
p.setName("John Doe");
name = p.getName();
System.out.println(name);
```

Output:

```
John Smith
John Doe
```

To understand the effects of **public** and **private**, consider the following program:

```
// Public vs private access.
```

```
class MyClass {
private int alpha; // private access
public int beta; // public access
int gamma; // default access (essentially public)
/* Methods to access alpha. It is OK for a
member of a class to access a private member of the same class.
*/
void setAlpha(int a) {
alpha = a;
}
int getAlpha() {
return alpha;
}
}

class AccessDemo {
public static void main(String args[]) {
MyClass ob = new MyClass();

/* Access to alpha is allowed only through
its accessor methods. */
ob.setAlpha(-99);
System.out.println("ob.alpha is " + ob.getAlpha());
// You cannot access alpha like this:
// ob.alpha = 10; // Wrong! alpha is private!
// These are OK because beta and gamma are public.
ob.beta = 88;
ob.gamma = 99;
}
}
```

As you can see, inside the **MyClass** class, **alpha** is specified as **private**, **beta** is explicitly specified as **public**, and **gamma** uses the default access, which for this example is the same as specifying **public**. Because **alpha** is private, it cannot be accessed by code outside of its class.

Therefore, inside the **AccessDemo** class, **alpha** cannot be used directly. It must be accessed through its public accessor methods: **setAlpha( )** and **getAlpha( )**. If you were to remove the comment symbol from the beginning of the following line,

```
// ob.alpha = 10; // Wrong! alpha is private!
```

You would not be able to compile this program because of the access violation. Although access to **alpha** by code outside of **MyClass** is not allowed, methods defined within **MyClass** can freely access it, as the **setAlpha( )** and **getAlpha( )** methods show. The key point is this: a private member can be used freely by other members of its class, but it cannot be accessed by code outside its class.